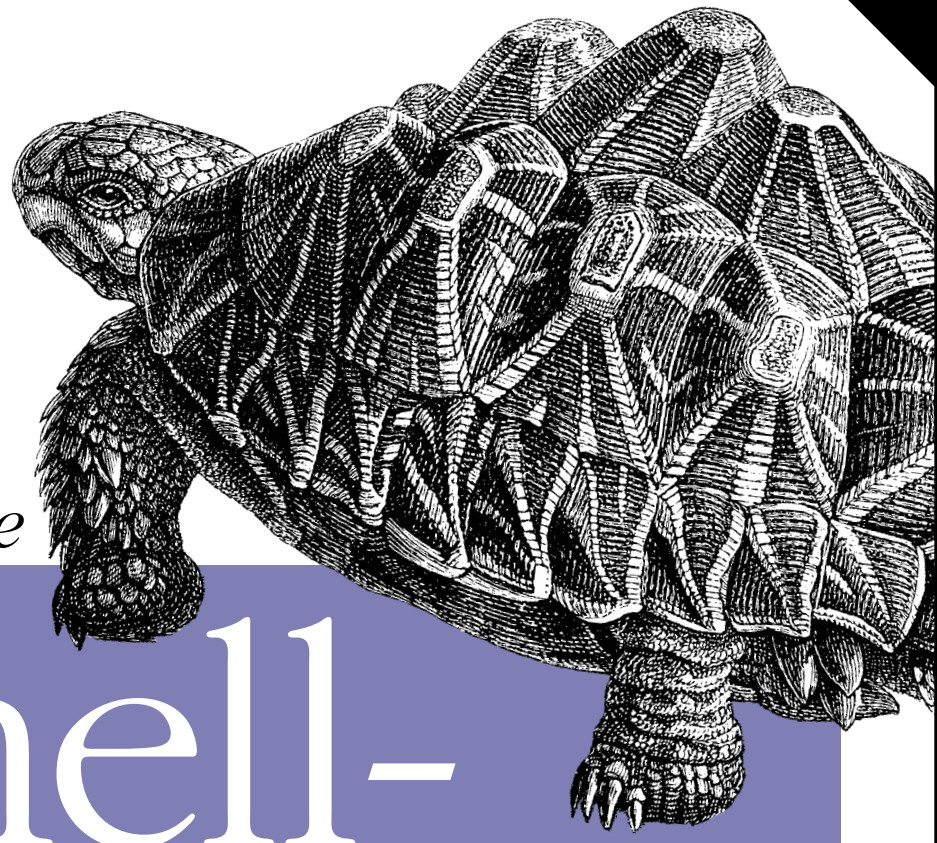
Automatisieren Sie Ihre Unix/Linux-Tasks

Deutsche Ausgabe



Klassische

Shell-Programmierung

Arnold Robbins & Nelson H. F. Beebe

Deutsche Übersetzung von Kathrin Lichtenberg

O'REILLY®

Klassische Shell-Programmierung

Arnold Robbins & Nelson H. F. Beebe

Deutsche Übersetzung von Kathrin Lichtenberg

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

Die Informationen in diesem Buch wurden mit größter Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Verlag, Autoren und Übersetzer übernehmen keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene Fehler und deren Folgen.
Alle Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt und sind möglicherweise eingetragene Warenzeichen. Der Verlag richtet sich im wesentlichen nach den Schreibweisen der Hersteller. Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Alle Rechte vorbehalten einschließlich der Vervielfältigung, Übersetzung, Mikroverfilmung sowie Einspeicherung und Verarbeitung in elektronischen Systemen.

Kommentare und Fragen können Sie gerne an uns richten:
O'Reilly Verlag
Balthasarstr. 81
50670 Köln
Tel.: 0221/9731600
Fax: 0221/9731608
E-Mail: kommentar@oreilly.de

Copyright der deutschen Ausgabe:
© 2006 by O'Reilly Verlag GmbH & Co. KG
1. Auflage 2006

Die Originalausgabe erschien 2005 unter dem Titel
Classic Shell Scripting im Verlag O'Reilly Media, Inc.

Die Darstellung einer Schildkröte im Zusammenhang mit dem Thema Shell-Programmierung ist ein Warenzeichen von O'Reilly Media, Inc.

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.ddb.de* abrufbar.

Übersetzung und deutsche Bearbeitung: Kathrin Lichtenberg
Lektorat: Volker Bombien, Köln
Korrektorat: Oliver Mosler, Köln
Satz: Tim Mergemeier, reemers publishing services gmbh, Krefeld; www.reemers.de
Umschlaggestaltung: Emma Colby & Michael Henri Oreal
Produktion: Andrea Miß, Köln
Belichtung, Druck und buchbinderische Verarbeitung:
Druckerei Kösel, Krugzell; www.koeselbuch.de

ISBN-10 3-89721-441-5
ISBN-13 978-3-89721-441-5

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhalt

Vorwort . . . . . **IX**

Einführung . . . . . **XI**

1 Hintergrund **1**
1.1 Unix-Geschichte 1
1.2 Prinzipien für Software-Werkzeuge 4
1.3 Zusammenfassung 7

2 Zum Einstieg **8**
2.1 Skriptsprachen im Vergleich mit kompilierten Sprachen 8
2.2 Wozu ein Shell-Skript einsetzen? 9
2.3 Ein einfaches Skript 9
2.4 Eigenständige Skripten: #! in der ersten Zeile 10
2.5 Grundlegende Shell-Konstrukte 13
2.6 Auf Shell-Skript-Argumente zugreifen 24
2.7 Einfache Überwachung der Ausführung 26
2.8 Internationalisierung und Lokalisierung 26
2.9 Zusammenfassung 30

3 Suchen und Ersetzen **32**
3.1 Nach Text suchen 32
3.2 Reguläre Ausdrücke 35
3.3 Mit Feldern arbeiten 60
3.4 Zusammenfassung 69

4 Werkzeuge zur Textverarbeitung . . . 71
4.1 Text sortieren . . . 71
4.2 Duplikate löschen . . . 80
4.3 Absätze umformatieren . . . 81
4.4 Zeilen, Wörter und Zeichen zählen . . . 82
4.5 Drucken . . . 83
4.6 Die erste und die letzte Zeile extrahieren . . . 88
4.7 Zusammenfassung . . . 91

5 Mit Pipelines erstaunliches leisten . . . 92
5.1 Daten aus strukturierten Textdateien extrahieren . . . 92
5.2 Strukturierte Daten für das Web . . . 100
5.3 Beim Kreuzworträtsel schummeln . . . 106
5.4 Wortlisten . . . 108
5.5 Tag-Listen . . . 111
5.6 Zusammenfassung . . . 114

6 Variablen, Entscheidungen treffen und Aktionen wiederholen . . . 116
6.1 Variablen und Arithmetik . . . 116
6.2 Exit-Status . . . 128
6.3 Die case-Anweisung . . . 137
6.4 Schleifen . . . 138
6.5 Funktionen . . . 144
6.6 Zusammenfassung . . . 147

7 Eingabe und Ausgabe, Dateien und Befehlsauswertung . . . 149
7.1 Standardeingabe, Standardausgabe und Standardfehlerausgabe . . . 149
7.2 Zeilen lesen mit read . . . 150
7.3 Mehr über Umleitungen . . . 152
7.4 Die vollständige Geschichte von printf . . . 157
7.5 Tilde-Erweiterung und Wildcards . . . 162
7.6 Befehlsersetzung . . . 166
7.7 Quotierung . . . 172
7.8 Auswertungsreihenfolge und eval . . . 173
7.9 Integrierte Befehle . . . 179
7.10 Zusammenfassung . . . 187

8 Produktionsskripten . . . 189
8.1 Pfadsuche . . . 189
8.2 Software-Builds automatisieren . . . 205
8.3 Zusammenfassung . . . 236

9 awk bis zum Abwinken . . . 237
9.1 Die awk-Kommandozeile . . . 238
9.2 Das awk-Programmiermodell . . . 239
9.3 Programm-Elemente . . . 240
9.4 Datensätze und Felder . . . 251
9.5 Muster und Aktionen . . . 254
9.6 Einzeilige Programme in awk . . . 256
9.7 Anweisungen . . . 259
9.8 Benutzerdefinierte Funktionen . . . 269
9.9 Stringfunktionen . . . 272
9.10 Numerische Funktionen . . . 281
9.11 Zusammenfassung . . . 283

10 Arbeiten mit Dateien . . . 284
10.1 Dateien auflisten . . . 284
10.2 Änderungsdaten mit touch aktualisieren . . . 290
10.3 Temporäre Dateien erzeugen und verwenden . . . 292
10.4 Dateien suchen . . . 297
10.5 Befehle ausführen: xargs . . . 313
10.6 Informationen über den Speicherplatz im Dateisystem . . . 314
10.7 Dateien vergleichen . . . 319
10.8 Zusammenfassung . . . 327

11 Ein ausführliches Beispiel: Das Zusammenführen von Benutzerdatenbanken . . . 329
11.1 Das Problem . . . 329
11.2 Die Passwortdateien . . . 330
11.3 Zusammenführen von Passwortdateien . . . 331
11.4 Ändern der Datei-Eigentümerschaft . . . 339
11.5 Andere Probleme aus dem wirklichen Leben . . . 343
11.6 Zusammenfassung . . . 345

12 Rechtschreibprüfung . . . 347
12.1 Das Programm spell . . . 347
12.2 Der Original-Unix-Prototyp zur Rechtschreibprüfung . . . 348
12.3 Verbesserungen mit ispell und aspell . . . 350
12.4 Eine Rechtschreibprüfung in awk . . . 353
12.5 Zusammenfassung . . . 374

13 Prozesse **375**
13.1 Prozesserzeugung 376
13.2 Prozessauflistung 377
13.3 Prozesssteuerung und -löschung 384
13.4 Verfolgung von Systemaufrufen 391
13.5 Prozess-Accounting 396
13.6 Verzögerte Ausführung von Prozessen 397
13.7 Das /proc-Dateisystem 403
13.8 Zusammenfassung 404

14 Fragen der Shell-Portabilität und Erweiterungen **406**
14.1 Stolpersteine 406
14.2 Der bash-Befehl shopt 410
14.3 Gebräuchliche Erweiterungen 414
14.4 Download-Informationen 428
14.5 Andere erweiterte Shells im Bourne-Stil 431
14.6 Shell-Versionen 431
14.7 Shell-Initialisierung und -Beendigung 432
14.8 Zusammenfassung 438

15 Sichere Shell-Skripten: Ein Einstieg **440**
15.1 Tipps für sichere Shell-Skripten 440
15.2 Eingeschränkte Shell 443
15.3 Trojanische Pferde 445
15.4 Setuid bei Shell-Skripten: Eine schlechte Idee 446
15.5 ksh93 und privilegierter Modus 448
15.6 Zusammenfassung 449

A Manpages schreiben **451**

B Dateien und Dateisysteme **466**

C Wichtige Unix-Befehle **505**

Bibliografie **511**

Glossar **517**

Index **545**

Vorwort

Natürlich schreibe ich nicht schon seit 30 Jahren Shell-Skripten?!? Wenn ich allerdings darüber nachdenke, glaube ich, dass ich es wohl doch schon so lange tue, obgleich ich klein angefangen habe. (Die frühen Unix-Shells, vor der Bourne-Shell, waren nach modernen Maßstäben sehr primitiv, und das Schreiben umfangreicher Skripten war schwierig. Glücklicherweise wurde es dann schnell besser.)

In den letzten Jahren wurde die Shell als Skriptsprache vernachlässigt und unterschätzt. Und obwohl es die erste Unix-Skriptsprache war, ist sie immer noch eine der besten. Ihre Kombination aus Erweiterbarkeit und Effizienz bleibt einmalig, und die Verbesserungen, die sie über die Jahre erfahren hat, lassen sie weiterhin im Vergleich mit anderen Skriptsprachen bestehen, die hochgejubelt wurden. Die grafischen Benutzerschnittstellen sind heutzutage gefragter als die Kommandozeilen-Shells, allerdings bilden Skriptsprachen häufig die Grundlagen für die schicken Bildschirmgrafiken. Und die Shell ist in dieser Rolle auch weiterhin unübertroffen.

Die Abhängigkeit der Shell davon, dass andere Programme die meiste Arbeit erledigen, ist einerseits eine Schwäche, andererseits aber auch eine Stärke: Sie erhalten die präzise Notation einer Skriptsprache sowie die Geschwindigkeit und Effizienz von Programmen, die in C (usw.) geschrieben sind. Die Verwendung einer verbreiteten, allgemein einsetzbaren Datenrepräsentation – Textzeilen – in einer großen (und erweiterbaren) Menge von Werkzeugen erlaubt es der Skriptsprache, die Werkzeuge in endlosen Kombinationen miteinander zu verbinden. Das Ergebnis sind größere Flexibilität und Leistungsfähigkeit, als sie monolithische Software-Pakete mit eingebauten Menüeinträgen für (angeblich) alles, was Sie brauchen, jemals erreichen werden. Der frühe Erfolg der Shell bei der Übernahme dieses Ansatzes bestärkte die Entwicklung der Unix-Philosophie im Erschaffen spezialisierter Werkzeuge, die jeweils nur eine Aufgabe übernehmen und zusammengesetzt werden, um die Arbeit zu erledigen. Die Philosophie wiederum regte Verbesserungen an der Shell an, um sie mehr Aufgaben auf diese Weise ausführen zu lassen.

Shell-Skripten haben außerdem den Vorteil gegenüber C-Programmen – und auch gegenüber einigen der anderen Skriptsprachen (keine Namen!) –, dass sie im Allgemeinen recht einfach geschrieben und modifiziert werden können. Selbst Leute, die keine C-Pro-

grammierer sind, wie ein Großteil der Systemadministratoren heutzutage, arbeiten gern mit Shell-Skripten. Deshalb ist das Schreiben von Shell-Skripten sehr wichtig beim Erweitern von Benutzerumgebungen und beim Anpassen von Software-Paketen.

Tatsächlich gibt es hier einen »Kreislauf der Reinkarnationen«, den ich schon bei mehreren Software-Projekten bemerkt habe. Das Projekt setzt einfache Shell-Skripten an Schlüsselpositionen ein, um es den Benutzern zu ermöglichen, Aspekte der Software anzupassen. Es ist jedoch für das *Projekt* selbst viel einfacher, Probleme zu lösen, indem in diesen Shell-Skripten gearbeitet wird, anstatt in dem umgebenden C-Code, so dass die Skripten mit der Zeit immer komplizierter werden. Schließlich sind sie so kompliziert, dass die Benutzer nicht mehr einfach so damit zurechtkommen (einige der Skripten, die wir im C-News-Projekt geschrieben haben, waren als Belastungstests für die Shells berüchtigt, ganz zu schweigen von den Benutzern!), und so muss für die Anpassung durch die Benutzer ein neuer Satz Skripten bereitgestellt werden...

Lange schon fehlte ein gutes Buch über das Schreiben von Shell-Skripten. Bücher über die Unix-Programmierumgebung streifen das Thema, allerdings normalerweise nur kurz als eines von mehreren Themen, und die besseren Bücher waren schon lange überholt. Es gibt Referenzdokumentationen für die verschiedenen Shells, allerdings keine anfängerfreundliche Anleitung, die sowohl die Werkzeuge als auch die Shell behandelt, die Konzepte vorstellt, Hinweise gibt, wie man die besten Ergebnisse erzielen kann, und sich mit praktischen Problemen wie der Lesbarkeit befasst. Das gewünschte Buch sollte darüber hinaus auf die Unterschiede zwischen den verschiedenen Shells eingehen, anstatt vorzugeben, dass nur eine existiert.

Dieses Buch bietet all das und noch mehr. Hier ist endlich eine aktuelle und schmerzlose Einführung in die erste und beste der Unix-Skriptsprachen. Sie wird von realistischen Beispielen begleitet, die selbst wiederum nützliche Werkzeuge abgeben. Das Buch behandelt die Standard-Unix-Werkzeuge ausführlich genug, um Sie mit ihnen vertraut zu machen (und um eine nützliche Referenz für all diejenigen zu bieten, die von den Manpages abgeschreckt werden). Ich darf mit besonderer Freude feststellen, dass auch die Grundlagen von `awk` besprochen werden, einem äußerst nützlichen und leider vernachlässigten Werkzeug, das hervorragend dazu geeignet ist, die Lücken zwischen anderen Werkzeugen zu überbrücken und kleine Programmieraufgaben leicht und präzise zu erledigen.

Ich empfehle dieses Buch jedem, der Shell-Skripten verfasst oder von Unix abgeleitete Systeme administriert. Ich habe viel daraus gelernt und denke, Sie werden das auch tun.

– Henry Spencer
SP Systems

Einführung

Benutzer oder Programmierer, die neu bei Unix[1] sind, werden plötzlich mit einer verwirrenden Vielfalt von Programmen konfrontiert, die jeweils oft mehrere Optionen besitzen. Dabei drängen sich Fragen auf wie »Welchem Zweck dienen sie?« und »Wie benutze ich sie?«.

Die Aufgabe dieses Buches besteht darin, diese Fragen zu beantworten. Es lehrt Sie, wie Sie die Unix-Werkzeuge mit der Standard-Shell kombinieren, um Ihre Probleme zu lösen. Das ist die Kunst der *Shell-Skripten*. Das Schreiben von Shell-Skripten erfordert nicht nur Kenntnisse der Shell-Sprache, sondern auch Kenntnisse der einzelnen Unix-Programme: Weshalb gibt es jedes einzelne von ihnen und wie benutzt man sie eigenständig und in Kombination mit anderen Programmen?

Wieso sollten Sie das Schreiben von Shell-Skripten erlernen? Weil es oft möglich ist, mittelgroße bis große Probleme in kleinere Teile aufzuspalten, die jeweils mit einem der Unix-Werkzeuge gelöst werden können. Ein Shell-Skript kann – wenn es gut geschrieben ist – ein Problem in einem Bruchteil der Zeit lösen, die eine herkömmliche Programmiersprache wie C oder C++ dafür benötigen würde. Es ist auch möglich, Shell-Skripten *portabel* zu machen – d. h. nutzbar über einen großen Bereich von Unix- und POSIX-konformen Systemen – ohne großartige Anpassungen vornehmen zu müssen.

Wenn wir über Unix-Programme sprechen, benutzen wir absichtlich den Begriff *Werkzeuge*. Der Unix-*Werkzeugkasten-Ansatz* zur Lösung von Problemen ist bereits lange als »Software Werkzeuge«-Philosophie bekannt gewesen.[2]

Eine alte Analogie fasst diesen Ansatz zur Problemlösung zusammen. Ein Schweizer Taschenmesser ist so nützlich, dass viele Leute eines besitzen. Es hat mehrere Klingen, einen Schraubendreher, einen Dosenöffner, einen Zahnstocher usw. Größere Modelle

1 In diesem Buch meinen wir bei Verwendung des Begriffs *Unix* nicht nur kommerzielle Varianten des ursprünglichen Unix-Systems, wie etwa Solaris, Mac OS X und HP-UX, sondern auch frei verfügbare, ähnlich funktionierende Systeme wie GNU/Linux und die verschiedenen BSD-Systeme: BSD/OS, NetBSD, FreeBSD und OpenBSD.

2 Dieser Ansatz wurde durch das Buch *Software Tools* (Addison-Wesley) bekannt gemacht.

enthalten noch mehr Werkzeuge, wie etwa einen Korkenzieher oder eine Lupe. Es gibt jedoch gar nicht so viele Dinge, die Sie mit einem Schweizer Taschenmesser tun können. Es eignet sich zwar hervorragend zum Schnitzen oder einfachen Einkerben, Sie würden es aber niemals dazu verwenden, um eine Hundehütte oder ein Vogelhaus zu bauen. Stattdessen würden Sie auf *Spezialwerkzeuge* zurückgreifen, wie etwa Hammer, Säge, Zwinge oder Hobel. Wenn Sie Programmierprobleme lösen, ist es daher auch besser, spezialisierte Software-Werkzeuge zu verwenden.

Unsere Zielgruppe

Dieses Buch ist für Computer-Benutzer und Software-Entwickler gedacht, die sich in einer Unix-Umgebung wiederfinden und dort Shell-Skripten schreiben müssen. Beispielsweise könnten Sie ein Informatik-Student sein, der seinen ersten Zugang auf dem Unix-System seiner Universität hat. Sie wollen mehr über die Dinge lernen, die unter Unix möglich sind und die Ihr Windows-PC nicht annähernd kann. (In einem solchen Fall werden Sie wahrscheinlich Skripten schreiben, um Ihre Umgebung anzupassen.) Oder vielleicht haben Sie gerade angefangen, als Systemadministrator zu arbeiten, und müssen spezialisierte Programme für Ihr Unternehmen oder Ihre Hochschule schreiben. (Dazu fallen einem Protokollverwaltung und Abrechnung bzw. Accounting ein.) Sie sind möglicherweise ein erfahrener Mac OS-Entwickler, der nun in die schöne neue Welt von Mac OS X einsteigt, wo Installationsprogramme als Shell-Skripten geschrieben werden. Wer auch immer Sie sind, falls Sie mehr über Shell-Skripten lernen wollen, ist dieses Buch genau richtig für Sie. In diesem Buch lernen Sie:

Entwurfskonzepte und Prinzipien für Software-Werkzeuge
: Eine Reihe von Prinzipien wacht über den Entwurf und die Implementierung guter Software-Werkzeuge. Wir erläutern Ihnen diese Prinzipien und zeigen sie Ihnen in diesem Buch in Aktion.

Welches die Unix-Werkzeuge sind
: Beim Schreiben von Shell-Skripten werden bestimmte Unix-Werkzeuge immer wieder verwendet. Wir behandeln die Grundlagen der Shell sowie die der regulären Ausdrücke und präsentieren jedes der Werkzeuge im Kontext einer bestimmten Art von Problem. Neben der Funktionsweise der einzelnen Werkzeuge zeigen wir Ihnen jeweils, *weshalb* diese existieren und wieso sie bestimmte Optionen besitzen.

  Einführung in Unix ist – wie der Name schon sagt – eine Einführung in Unix-Systeme, die als Einstieg dient, um jemanden ohne Unix-Erfahrungen an Unix heranzuführen. Im Gegensatz dazu behandelt *Unix in a Nutshell* die ganze Vielfalt der Unix-Hilfsprogramme, bietet dabei aber kaum bzw. keine Hinweise, wann und wie ein bestimmtes Werkzeug benutzt wird. Unser Ziel besteht darin, die Lücke zwischen diesen beiden Büchern zu überbrücken: Wir zeigen Ihnen, wie Sie die Möglichkeiten entdecken, die Ihr Unix-System bietet, um Ihre Aufgaben schnell, effektiv und (hoffentlich) elegant zu erledigen.

Wie Sie die Werkzeuge kombinieren, um Ihre Aufgaben zu erfüllen
: Bei der Arbeit mit Shell-Skripten gilt tatsächlich, dass das »Ganze mehr ist als die Summe seiner Teile«. Indem Sie die Shell als »Leim« verwenden, um die einzelnen Werkzeuge zu kombinieren, können Sie mit geringen Anstrengungen Erstaunliches erreichen.

Mehr über beliebte Erweiterungen von Standardwerkzeugen
: Falls Sie ein GNU/Linux-System oder ein von BSD abgeleitetes System verwenden, ist es ziemlich wahrscheinlich, dass Ihre Werkzeuge über zusätzliche nützliche Funktionen und/oder Optionen verfügen. Wir behandeln diese ebenfalls.

Mehr über unentbehrliche Nichtstandardwerkzeuge
: Manche Programme gehören auf den traditionellen Unix-Systemen nicht zum »Standard«, sind aber dennoch zu nützlich, um ohne sie auszukommen. Wo es passend erscheint, werden sie ebenfalls behandelt. Außerdem verraten wir Ihnen, woher Sie diese Werkzeuge bekommen können.

Für Entwickler und Administratoren, die schon lange mit Unix arbeiten, ist die Philosophie der Software-Werkzeuge nichts Neues. Allerdings sind die Bücher, die diesen Ansatz bekannt gemacht haben, zum Großteil schon mehr als 20 Jahre alt, obwohl es sich immer noch lohnt, sie zu lesen! Unix-Systeme haben sich seit der Zeit, als diese Bücher geschrieben wurden, auf vielfältige Weise geändert. Wir hatten daher das Gefühl, dass es Zeit für eine aktualisierte Präsentation dieser Ideen ist, wobei wir für unsere Beispiele moderne Versionen der Werkzeuge und aktuelle Systeme einsetzen. Hier sind die Kernpunkte unseres Ansatzes:

- Unsere Darstellung beruht auf POSIX. »POSIX« ist der Kurzname einer Reihe von formalen Standards, die auf programmatischer Ebene (C, C++, Ada, Fortran) sowie auf der Ebene der Shell und der Hilfsprogramme eine portable Betriebssystemumgebung beschreiben. Die POSIX-Standards waren sehr erfolgreich dabei, Entwicklern eine Möglichkeit an die Hand zu geben, sowohl ihre Programme als auch ihre Shell-Skripten für einen großen Bereich von Systemen unterschiedlicher Hersteller portabel zu gestalten. Wir präsentieren die Shell-Sprache und alle Werkzeuge mit ihren nützlichsten Optionen so, wie sie im neuesten POSIX-Standard beschrieben werden.

  Der offizielle Name des Standards lautet IEEE Std. 1003.1–2001.[3] Dieser Standard enthält mehrere optionale Teile, von denen der wichtigste die *X/Open System Interface*-Spezifikationen (XSI) sind. Diese Funktionen dokumentieren einen vollen Bereich des Verhaltens des historischen Unix-Systems. Wo es wichtig ist, werden wir Änderungen zwischen dem aktuellen Standard und dem früheren Standard aus

3 Die Ausgabe aus dem Jahre 2004 wurde veröffentlicht, nachdem der (amerikanische) Text für dieses Buch abgeschlossen worden war. Falls Ihr Hauptziel darin besteht, das Schreiben von Shell-Skripten zu erlernen, spielen die Unterschiede zwischen den Standards von 2001 und 2004 keine Rolle.

dem Jahre 1992 erwähnen. Außerdem nennen wir die Funktionen, die mit XSI zu tun haben. Ein guter Ausgangspunkt für mit Unix zusammenhängende Standards ist *http://www.unix.org/*.[4]

Die Homepage der Single UNIX Specification lautet *http://www.unix.org/version3/*. Der Online-Zugriff auf den aktuellen Standard ist möglich, erfordert aber eine Registrierung unter *http://www.unix.org/version3/online.html*.

Gelegentlich bezeichnet der Standard ein bestimmtes Verhalten als »unspecified« oder nicht spezifiziert. Das geschieht mit Absicht, um es Herstellern zu erlauben, historisches Verhalten als *Erweiterung* zu unterstützen, das sind zusätzliche Features, die über das hinausgehen, was im Standard selbst dokumentiert ist.

- Wir wollen Ihnen nicht nur sagen, wie Sie ein bestimmtes Programm ausführen sollen, sondern auch darstellen, *wieso* das Programm existiert und welches Problem es löst. Indem Sie erfahren, weshalb ein Programm geschrieben wurde, können Sie besser verstehen, wann und wie Sie es verwenden sollen.
- Viele Unix-Programme verfügen über eine verblüffende Anzahl von Optionen. Normalerweise sind einige dieser Optionen für die tägliche Arbeit an den Problemen besser geeignet als andere. Wir sagen Ihnen für jedes Programm, welche Optionen am nützlichsten sind. Um genau zu sein, behandeln wir nicht alle Optionen, die die einzelnen Programme haben, sondern überlassen diese Aufgabe der Manpage des Programms oder anderen Referenzbüchern wie *Unix in a Nutshell* (O'Reilly) und *Linux in a Nutshell* (O'Reilly).

Sobald Sie dieses Buch beendet haben, sollten Sie nicht nur die Unix-Werkzeuge verstanden, sondern auch die Unix-Mentalität und die Philosophie der Software-Werkzeuge verinnerlicht haben.

Was Sie bereits wissen sollten

Folgende Dinge sollten Sie bereits wissen:

- Wie Sie sich an Ihrem Unix-System anmelden
- Wie Sie Programme auf der Kommandozeile ausführen
- Wie Sie einfache Pipelines aus Befehlen erstellen und einfache Eingabe-/Ausgabeumleitungen wie < und > benutzen
- Wie Sie Jobs mit `&` in den Hintergrund verschieben
- Wie Sie Dateien anlegen und bearbeiten
- Wie Sie Skripten mit `chmod` ausführbar machen

4 Eine Datei mit häufig gestellten technischen Fragen (FAQ) über IEEE Std. 1003.1–2001 finden Sie unter *http://www.opengroup.org/austin/papers/posix_faq.html*. Einige Hintergrundinformationen über den Standard gibt es unter *http://www.opengroup.org/austin/papers/backgrounder.html*.

Falls Sie außerdem versuchen, die Beispiele hier durchzuarbeiten, indem Sie Befehle in Ihr Terminal (oder wahrscheinlicher: Ihre Terminal-Emulation) eintippen, empfehlen wir Ihnen die Benutzung einer POSIX-konformen Shell wie etwa eine neuere Version von `ksh93` oder die aktuelle Version der `bash`. Vor allem `/bin/sh` ist auf kommerziellen Unix-Systemen möglicherweise nicht vollständig POSIX-konform.

Kapitel 14 enthält URLs zum Herunterladen von `ksh93`, `bash` und `zsh` aus dem Internet.

Zusammenfassung der einzelnen Kapitel

Wir empfehlen Ihnen, das Buch in der angegebenen Reihenfolge zu lesen, da die einzelnen Kapitel jeweils auf den Konzepten und Materialien aufbauen, die in den vorhergehenden Kapiteln behandelt wurden. Hier ist eine kapitelweise Zusammenfassung:

Kapitel 1, *Hintergrund*
: Hier stellen wir kurz die Geschichte von Unix vor. Vor allem gehen wir auf die spezielle Umgebung an den Bell Labs ein, wo Unix entwickelt wurde und einen Großteil der Software Werkzeuge-Philosophie anregte. Dieses Kapitel präsentiert außerdem die Prinzipien für gute Software-Werkzeuge, die dann im Rest des Buches veranschaulicht werden.

Kapitel 2, *Zum Einstieg*
: Mit diesem Kapitel beginnt die eigentliche Besprechung. Es fängt damit an, dass kompilierte Sprachen und Skriptsprachen sowie die Unterschiede zwischen beiden beschrieben werden. Anschließend werden die Grundlagen der Shell-Skripten anhand zweier einfacher, aber nützlicher Beispiele vorgestellt. Folgende Themen werden besprochen: Befehle, Optionen, Argumente, Shell-Variablen, die Ausgabe mit `echo` und `printf`, einfache Ein-/Ausgabeumleitung, Befehlssuche, Zugriff auf Argumente aus einem Skript heraus und die Ausführungsverfolgung. Das Kapitel schließt mit einem Blick auf Internationalisierung und Lokalisierung; Fragen, die im »globalen Dorf« heutzutage immer wichtiger werden.

Kapitel 3, *Suchen und Ersetzen*
: Hier stellen wir die Textsuche (oder »Textfilterung«) mit regulären Ausdrücken vor. Wir behandeln außerdem die Durchführung von Änderungen und das Extrahieren von Text. Das sind wichtige Operationen, die die Grundlage eines Großteils der Shell-Skripten bilden.

Kapitel 4, *Werkzeuge zur Textverarbeitung*
: In diesem Kapitel beschreiben wir eine Reihe von Textverarbeitungswerkzeugen, die in Shell-Skripten immer wieder eingesetzt werden. Zwei der wichtigsten Werkzeuge, die hier vorgestellt werden, sind `sort` und `uniq`, die leistungsfähige Methoden zum Organisieren und Reduzieren von Daten sind. Dieses Kapitel beschreibt außerdem das Neuformatieren von Absätzen, das Zählen von Texteinheiten, das Drucken von Dateien und das Abrufen der ersten oder letzten Zeilen einer Datei.

Kapitel 5, *Mit Pipelines Erstaunliches leisten*
: Dieses Kapitel zeigt mehrere kleine Skripten, die das Kombinieren einfacher Unix-Hilfsprogramme demonstrieren, um damit leistungsfähigere und – was noch wichtiger ist – flexiblere Werkzeuge herzustellen. Im Prinzip ist dieses Kapitel ein Kochbuch, das Probleme und Rezepte zu deren Lösung präsentiert, wobei das gemeinsame Thema darin besteht, dass die Lösungen aus linearen Pipelines aufgebaut werden.

Kapitel 6, *Variablen, Entscheidungen treffen und Aktionen wiederholen*
: Das ist das erste der beiden Kapitel, in denen die restlichen Grundlagen der Shell-Sprache präsentiert werden. Dieses Kapitel betrachtet Shell-Variablen und die Shell-Arithmetik, das wichtige Konzept eines Exit-Status sowie die Frage, wie in der Shell Entscheidungen getroffen und Schleifen aufgebaut werden. Es wird mit einer Besprechung der Shell-Funktionen abgeschlossen.

Kapitel 7, *Eingabe und Ausgabe, Dateien und Befehlsauswertung*
: Dieses Kapitel vervollständigt die Beschreibung der Shell. Es konzentriert sich dabei auf die Eingabe und Ausgabe, die verschiedenen Ersetzungen, die die Shell durchführt, die Quotierung, die Auswertungsreihenfolge der Kommandozeile und integrierte Shell-Befehle.

Kapitel 8, *Produktionsskripten*
: Hier zeigen wir Kombinationen aus Unix-Werkzeugen, um damit komplexere Textverarbeitungsaufgaben durchzuführen. Die Programme in diesem Kapitel sind umfangreicher als diejenigen in Kapitel 5, aber immer noch kurz genug, um in einigen Minuten durchschaut zu werden. Dennoch erledigen sie Aufgaben, die mit herkömmlichen Programmiersprachen wie C, C++ oder Java™ nur schwer durchzuführen wären.

Kapitel 9, *awk bis zum Abwinken*
: Dieses Kapitel beschreibt die Grundlagen der Sprache `awk`. `awk` ist selbst eine leistungsfähige Sprache. Einfache und manchmal auch nicht so einfache `awk`-Programme können jedoch zusammen mit anderen Programmen in der Software-Werkzeugsammlung verwendet werden, um Daten zu extrahieren, zu bearbeiten und zu formatieren.

Kapitel 10, *Arbeiten mit Dateien*
: Dieses Kapitel stellt die wichtigsten Werkzeuge für das Arbeiten mit Dateien vor. Es behandelt das Auflisten von Dateien, das Erzeugen temporärer Dateien und den ausgesprochen wichtigen `find`-Befehl zum Suchen von Dateien, die bestimmten Kriterien entsprechen. Es betrachtet zwei wichtige Befehle zur Ermittlung der Auslastung des Festplattenplatzes und erläutert anschließend verschiedene Programme zum Vergleichen von Dateien.

Kapitel 11, *Ein ausführliches Beispiel: Das Zusammenführen von Benutzerdatenbanken*
: Hier verknüpfen wir alle bisher gewonnenen Erkenntnisse, indem wir eine interessante und halbwegs anspruchsvolle Aufgabe lösen.

Kapitel 12, *Rechtschreibprüfung*

In diesem Kapitel wird anhand des Problems der Rechtschreibprüfung gezeigt, wie man diese Aufgabe auf unterschiedliche Weise lösen kann. Die Original-Shell-Skript-Pipeline sowie zwei kleine Skripten werden vorgestellt, um die frei verfügbaren Befehle `ispell` und `aspell` für eine Rechtschreibprüfung als Stapelverarbeitung nutzbar zu machen. Das Kapitel schließt mit einem einigermaßen überschaubaren, aber dennoch leistungsfähigen Programm zur Rechtschreibprüfung in `awk`, das die Eleganz dieser Sprache auf ansprechende Weise demonstriert.

Kapitel 13, *Prozesse*

In diesem Kapitel verlassen wir den Bereich der Textverarbeitung und wenden uns der Job- und Systemverwaltung zu. Es gibt eine kleine Anzahl wichtiger Hilfsprogramme für die Verwaltung von Prozessen. Darüber hinaus behandelt dieses Kapitel den Befehl `sleep`, der für Skripten nützlich ist, die darauf warten, dass etwas passiert, sowie die anderen Standardwerkzeuge für die verzögerte oder zu einem festen Zeitpunkt des Tages stattfindende Befehlsverarbeitung. Was noch wichtiger ist, das Kapitel behandelt auch den `trap`-Befehl, der Shell-Skripten die Kontrolle über Unix-Signale gibt.

Kapitel 14, *Fragen der Shell-Portabilität und Erweiterungen*

Hier beschreiben wir einige der nützlicheren Erweiterungen, die sowohl in `ksh` als auch in `bash` zur Verfügung stehen, aber nicht Teil von POSIX sind. In vielen Fällen können Sie diese Erweiterungen sicher in Ihren Skripten verwenden. Wir betrachten außerdem eine Reihe von »Fallen«, die nur darauf warten, den unbedarften Shell-Skript-Autor zu erwischen. Es geht um Fragen, die auftreten, wenn Sie Skripten schreiben, sowie um mögliche Abweichungen zwischen den Implementierungen. Darüber hinaus werden Download- und Kompilierungsinformationen für `ksh` und `bash` geliefert. Das Kapitel endet mit einer Besprechung der Shell-Initialisierung und -Beendigung, die sich auf den unterschiedlichen Shell-Implementierungen unterscheiden.

Kapitel 15, *Sichere Shell-Skripten: Ein Einstieg*

In diesem Kapitel geben wir eine oberflächliche Einführung in Sicherheitsfragen bezüglich Shell-Skripten.

Anhang A, *Manpages schreiben*

Dieses Kapitel erläutert, wie man eine Manpage schreibt. Diese notwendige Fertigkeit wird in typischen Unix-Büchern normalerweise vernachlässigt.

Anhang B, *Dateien und Dateisysteme*

Hier beschreiben wir das Unix-Bytestrom-Dateisystemmodell, das wir komplexeren historischen Dateisystemen gegenüberstellen. Wir erläutern, weshalb seine Einfachheit ein Vorzug ist.

Anhang C, *Wichtige Unix-Befehle*

Dieses Kapitel enthält mehrere Listen mit Unix-Befehlen. Sie sollten diese Befehle und ihre Funktionsweise lernen, um Ihre Fähigkeiten als Unix-Entwickler zu verbessern.

Bibliografie
: Hier führen wir weitere Informationsquellen zum Schreiben von Shell-Skripten mit Unix auf.

Glossar
: Das Glossar bietet Definitionen für die wichtigen Begriffe und Konzepte, die in diesem Buch eingeführt wurden.

In diesem Buch verwendete Konventionen

Wir setzen es als selbstverständlich voraus, dass Ihnen bekannt ist, dass Sie nach dem Eingeben eines Shell-Befehls die Enter-Taste drücken müssen.

Zeichen, die mit Strg-*X* angegeben werden, wobei *X* ein beliebiger Buchstabe ist, werden eingegeben, indem man die Strg-Taste drückt und hält und dann den angegebenen Buchstaben drückt. Wir geben den Buchstaben zwar in Großschreibung an, dennoch drücken Sie die Taste ohne gleichzeitiges Halten der Umschalt- (oder Shift-) Taste.

Weitere Sonderzeichen sind Newline (identisch mit Strg-J), Backspace (identisch mit Strg-H), Esc(ape), Tab(ulator) und Entf(ernen).

In diesem Buch kommen folgende Schriftkonventionen zum Einsatz:

Kursivschrift
: *Kursivschrift* wird im Text für Betonungen verwendet, um besondere Begriffe hervorzuheben, die erstmalig definiert werden, für E-Mail-Adressen und Internet-URLs sowie für Zitate aus Manpages. Sie wird außerdem benutzt, wenn Dummy-Parameter besprochen werden, die durch einen tatsächlichen Wert ersetzt werden sollen, und um Kommentare in Beispielen anzugeben.

`Nichtproportionalschrift`
: Diese Schrift wird für Unix-Dateinamen, externe und integrierte Befehle sowie Befehlsoptionen verwendet. Sie kommt außerdem für Variablennamen und Shell-Schlüsselwörter, -Optionen und -Funktionen, für Dateinamensuffixe und in Beispielen für den Inhalt von Dateien oder die Ausgaben von Befehlen sowie für Kommandozeilen oder Beispieleingaben innerhalb des normalen Textes zum Einsatz. Kurz gesagt, steht alles, was mit der Computer-Benutzung zu tun hat, in dieser Schrift.

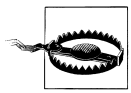
`Nichtproportionalschrift fett`
: Diese Schrift wird im Text verwendet, um reguläre Ausdrücke und Shell-Wildcard-Muster von dem zu filternden Text zu unterscheiden. Sie wird außerdem in Beispielen eingesetzt, um die Interaktion zwischen dem Benutzer und der Shell zu zeigen; alles, was der Benutzer eintippt, wird in **`fett gedruckter Nichtproportionalschrift`** angezeigt. Zum Beispiel:

```
$ pwd                          Das hat der Benutzer eingetippt
/home/tolstoy/novels/w+p       Das hat das System ausgegeben
$
```

Nichtproportionalschrift kursiv
: Diese Schrift wird im Text und in Beispielkommandozeilen für Dummy-Parameter verwendet, die durch einen tatsächlichen Wert ersetzt werden sollen. Zum Beispiel:

```
$ cd verzeichnis
```

Dieses Symbol kennzeichnet einen Hinweis, einen Vorschlag oder eine allgemeine Anmerkung.



Dieses Symbol kennzeichnet eine Warnung.

Verweise auf Einträge im Unix User's Manual werden im Standardstil geschrieben: *name*(N), wobei *name* der Befehlsname ist und N die Nummer des Abschnitts (üblicherweise 1), in dem die Informationen gefunden werden können. Beispielsweise bedeutet *grep*(1) die Manpage für `grep` in Abschnitt 1. Die Referenzdokumentation wird kurz als »Manpage« (oder Manual Page) bezeichnet.

Wir bezeichnen sowohl Unix-Systemaufrufe als auch C-Bibliotheksfunktionen folgendermaßen: `open( )`, `printf( )`. Sie erhalten die Manpage für beide Arten von Aufrufen, indem Sie den Befehl `man` verwenden:

```
$ man open          Die Manpage open(2) anschauen
$ man printf        Die Manpage printf(3) anschauen
```

Wenn Programme eingeführt werden, dann beschreibt ein Kasten wie der folgende sowohl das Werkzeug als auch wichtige Optionen, Verwendungshinweise und die Aufgabe des Werkzeugs.

Code-Beispiele

Dieses Buch ist voll von Beispielen für Shell-Befehle und Programme, die nicht nur die erläuterte Eigenschaft verdeutlichen sollen, sondern dazu gedacht sind, Ihr tägliches Leben als Benutzer oder Programmierer zu vereinfachen. Wir ermuntern Sie insbesondere, die Beispiele selbst zu verändern und zu erweitern.

Der Code in diesem Buch wird unter den Bedingungen der GNU General Public License (GPL) veröffentlicht, die das Kopieren, Verwenden und Verändern dieser Programme erlaubt. Den exakten Wortlaut der Lizenz finden Sie in der Datei `COPYING`, die den Beispielen beigegeben ist.

Der Code steht auf der Website des Buchs zur Verfügung:
http://www.oreilly.com/catalog/shellsrptg/index.html.

Beispiel

Verwendung
: `whizprog` [*optionen* ...] [*argumente* ...]
Dieser Abschnitt zeigt Ihnen, wie Sie den Befehl, der hier mit `whizprog` bezeichnet ist, ausführen.

Zweck
: Dieser Abschnitt beschreibt, *wozu* das Programm existiert.

Wichtige Optionen
: Dieser Abschnitt führt die Optionen auf, die für den täglichen Einsatz des besprochenen Programms wichtig sind.

Verhalten
: Dieser Abschnitt fasst zusammen, was das Programm tut.

Warnungen
: Wenn es etwas zu beachten gilt, wird es hier erwähnt.

Wir begrüßen es, erwähnt zu werden, verlangen es jedoch nicht. Zu einer Erwähnung gehören normalerweise Titel, Autor, Verlag und ISBN. Zum Beispiel: »*Klassische Shell-Programmierung* von Arnold Robbins und Nelson H. F. Beebe. Copyright 2006 O'Reilly Verlag, ISBN 3-89721-441-5«.

Unix-Werkzeuge für Windows-Systeme

Viele Programmierer, die ihre ersten Erfahrungen auf Unix-Systemen gesammelt haben und nachfolgend in die PC-Welt gewechselt sind, haben sich eine hübsche Unix-artige Umgebung gewünscht (vor allem, wenn sie mit der grauenhaften MS-DOS-Kommandozeile konfrontiert wurden!). Es ist daher nicht überraschend, dass es inzwischen mehrere Schnittstellen im Stil der Unix-Shell für PC-Betriebssysteme gibt.

In den vergangenen Jahren sind uns nicht nur Shell-Klone, sondern auch ganze Unix-Umgebungen begegnet. Zwei von ihnen verwenden `bash` und `ksh93`. Eine andere bietet ihre eigene Shell-Neuimplementierung. Dieser Abschnitt beschreibt alle Umgebungen (in alphabetischer Reihenfolge) und gibt Kontakt- und Download-Informationen an.

Cygwin

Cygnus Consulting (inzwischen Red Hat) schuf die `cygwin`-Umgebung. Nachdem in dem Unternehmen zuerst `cygwin.dll`, eine Shared Library, geschaffen wurde, die eine Emulation von Unix-Systemaufrufen zur Verfügung stellt, portierte man dort eine große Anzahl von Unix-Hilfsprogrammen auf die verschiedenen Versionen von Microsoft Windows. Die Emulation schließt TCP/IP-Netzwerktechniken mit der Berkeley-Socket-API ein. Die

größte Funktionalität gibt es unter Windows/NT, Windows 2000 und Windows XP, obwohl die Umgebung auch unter Windows 95/98/ME funktioniert.

Die `cygwin`-Umgebung verwendet `bash` als Shell, GCC als C-Compiler und den Rest der GNU-Hilfsprogramme als Unix-Werkzeugsammlung. Ein ausgeklügelter `mount`-Befehl ermöglicht eine Abbildung der Windows-`C:\Pfad`-Notation auf Unix-Dateinamen.

Der Ausgangspunkt für das `cygwin`-Projekt ist *http://www.cygwin.com/*. Als Erstes müssen Sie ein Installationsprogramm herunterladen. Wenn Sie es ausführen, entscheiden Sie, welche zusätzlichen Pakete Sie installieren wollen. Die Installation verläuft vollständig Internet-basiert, es gibt keine offiziellen `cygwin`-CDs, zumindest nicht von den Betreuern des Projekts.

DJGPP

Das DJGPP-Paket bietet 32-Bit-GNU-Werkzeuge für die MS-DOS-Umgebung. Um die Webseite zu zitieren:

> DJGPP is a complete 32-bit C/C++ development system for Intel 80386 (and higher) PCs running MS-DOS. It includes ports of many GNU development utilities. The development tools require an 80386 or newer computer to run, as do the programs they produce. In most cases, the programs it produces can be sold commercially without license or royalties.[5]

Der Name stammt von den Initialen von D.J. Delorie, der den GNU C++-Compiler, `g++`, nach MS-DOS portiert hat, und den Textinitialen von `g++`, GPP. Inzwischen ist aus dem System eine vollständige Unix-Umgebung geworden, die auf MS-DOS aufsetzt und alle GNU-Werkzeuge sowie die `bash` als Shell mitbringt. Im Gegensatz zu `cygwin` oder UWIN (siehe weiter unten) benötigen Sie keine Version von Windows, sondern nur einen 32-Bit-Prozessor und MS-DOS. (Obwohl Sie natürlich DJGPP in einem Windows-MS-DOS-Fenster benutzen können.) Die Website lautet *http://www.delorie.com/djgpp/*.

MKS Toolkit

Die vielleicht etablierteste Unix-Umgebung für die PC-Welt ist das MKS Toolkit von Mortice Kern Systems:

MKS Canada – Corporate Headquarters	1-519-884-2251
410 Albert Street	1-519-884-8861 (FAX)
Waterloo, ON	1-800-265-2797 (Sales)
Canada N2L 3V3	*http://www.mks.com/*

5 DJGPP ist ein vollständiges 32-Bit-C/C++-Entwicklungssystem für Intel 80386-PCs (und neuer), auf denen MS-DOS läuft. Es enthält Portierungen vieler GNU-Entwicklungsprogramme. Die Entwicklungswerkzeuge erfordern zum Ausführen einen 80386-Computer oder neuer, ebenso wie die Programme, die man mit ihnen herstellt. In den meisten Fällen können die Programme, die dieses System erzeugt, kommerziell ohne Lizenz oder Tantiemen verkauft werden.

Das MKS Toolkit gibt es in verschiedenen Versionen, je nach der Entwicklungsumgebung und der Anzahl der Benutzer, die es einsetzen werden. Es enthält eine Shell, die dem POSIX-Standard folgt, sowie praktisch alle Eigenschaften und Funktionen der Korn Shell von 1988 und mehr als 300 Hilfsprogramme wie etwa `awk`, `perl`, `vi`, `make` usw. Die MKS-Bibliothek unterstützt mehr als 1500 Unix-APIs, was das Ganze zu einer nahezu vollständigen und hilfreichen Portierung in die Windows-Umgebung macht.

AT&T UWIN

Das UWIN-Paket ist ein Projekt von David Korn und seinen Kollegen, mit dem unter Microsoft Windows eine Unix-Umgebung zur Verfügung gestellt werden soll. Es ähnelt in seiner Struktur dem bereits besprochenen `cygwin`. Eine Shared Library, `posix.dll`, bietet eine Emulation der Unix-Systemaufruf-APIs. Die Systemaufruf-Emulation ist fast vollständig. Ein interessanter Dreh ist, dass auf die Windows-Registry als Dateisystem unter `/reg` zugegriffen werden kann. Auf der Unix-API-Emulation wurden `ksh93` und mehr als 200 Unix-Hilfsprogramme (oder besser Neuimplementierungen) kompiliert und ausgeführt. Die UWIN-Umgebung baut auf dem nativen Microsoft Visual C/C++-Compiler auf, obwohl auch die GNU-Entwicklungswerkzeuge zum Herunterladen und Verwenden mit UWIN verfügbar sind.

http://www.research.att.com/sw/tools/uwin/ ist die Webseite des Projekts. Sie beschreibt, was es alles gibt, und bietet sowohl Links zum Herunterladen der Binärpakete als auch Informationen für die kommerzielle Lizenzierung des UWIN-Pakets. Es sind außerdem Links auf verschiedene Artikel über UWIN, zusätzliche nützliche Software und ähnliche Pakete enthalten.

Der bemerkenswerteste Vorteil des UWIN-Pakets besteht darin, dass seine Shell die authentische `ksh93` *ist*. Das bedeutet, dass die Kompatibilität zur Unix-Version von `ksh93` kein Problem darstellt.

Danksagungen

Jeder von uns möchte dem anderen für seine Bemühungen danken. In Anbetracht der Tatsache, dass wir uns nie persönlich begegnet sind, hat die Zusammenarbeit ziemlich gut funktioniert. Wir beide danken unseren Frauen auf das Herzlichste für ihre Mithilfe, Geduld, Liebe und Unterstützung während der Arbeit an diesem Buch.

Chet Ramey, der Betreuer der `bash`, beantwortete unzählige Fragen über die Feinheiten der POSIX-Shell. Glenn Fowler und David Korn von AT&T Research sowie Jim Meyering vom GNU Project beantworteten ebenfalls verschiedene Fragen. (In alphabetischer Reihenfolge) Keith Bostic, George Coulouris, Mary Ann Horton, Bill Joy, Rob Pike, Hugh Redelmeier (mit Hilfe von Henry Spencer) und Dennis Ritchie beantworteten Fragen zur Unix-Geschichte. Nat Torkington, Allison Randall und Tatiana Diaz von O'Reilly Media begleiteten das Buch von der Konzeption bis zur Fertigstellung. Robert Romano von O'Reilly machte aus unseren ursprünglichen ASCII- und `pic`-Skizzen her-

vorragende Abbildungen. Angela Howard erzeugte einen umfassenden Index für das Buch, der unseren Lesern von großem Nutzen sein sollte.

Geoff Collyer, Robert Day, Leroy Eide, John Halleck und Henry Spencer (in alphabetischer Reihenfolge) waren die technischen Gutachter für den ersten Entwurf dieses Buchs. Sean Burke begutachtete den zweiten Entwurf. Wir danken allen für ihre wertvollen und hilfreichen Beiträge.

Henry Spencer ist der Unix-Guru eines Unix-Gurus. Wir danken ihm für seine freundlichen Worte im Vorwort.

Der Zugang zu Unix-Systemen an der University of Utah in den Departments of Electrical and Computer Engineering, Mathematics, and Physics sowie am Center for High-Performance Computing wie auch die Gastzugänge, die freundlicherweise von IBM und Hewlett-Packard zur Verfügung gestellt wurden, waren wesentlich für das Testen der Software, die für dieses Buch geschrieben werden musste. Wir danken allen außerordentlich dafür.

– Arnold Robbins
– Nelson H.F. Beebe

KAPITEL 1

Hintergrund

Dieses Kapitel bietet einen kurzen Abriss der Entwicklung des Unix-Systems. Wenn Sie verstehen, wo und wie sich Unix entwickelt hat und mit welcher Absicht es entworfen wurde, können Sie auch seine Werkzeuge besser einsetzen. Dieses Kapitel stellt Ihnen außerdem die Prinzipien für Software-Werkzeuge vor, die dann im Rest des Buches demonstriert werden.

1.1 Unix-Geschichte

Wahrscheinlich wissen Sie bereits etwas über die Entwicklung von Unix. Viele Ressourcen berichten darüber hinaus über die komplette Geschichte. Wir wollen Ihnen hier zeigen, wie die Umgebung, der Unix entsprang, die Gestaltung der verschiedenen Werkzeuge beeinflusst hat.

Unix wurde ursprünglich im Computing Sciences Research Center der Bell Telephone Laboratories entwickelt.[1] Die erste Version entstand 1970, kurz nachdem die Bell Labs sich aus dem Multics-Projekt zurückzogen. Viele der Ideen, die Unix bekannt machten, stammten eigentlich aus dem Betriebssystem Multics; vor allem das Konzept von Geräten (Devices) als Dateien sowie das Vorhandensein eines Befehlsinterpreters (auch als *Shell* bezeichnet), der absichtlich nicht in das Betriebssystem integriert wurde. Eine gut geschriebene Historie (in englischer Sprache) finden Sie unter *http://www.bell-labs.com/history/unix*.

Da Unix in einer forschungsorientierten Umgebung entwickelt wurde, gab es auch keinen kommerziellen Druck, ein fertiges Produkt herzustellen oder zu verkaufen. Das hatte mehrere Vorteile:

- Das System wurde von seinen Benutzern entwickelt. Diese verwendeten es, um damit echte, tagtäglich auftretende Rechenprobleme zu lösen.

1 Der Name hat sich seitdem wenigstens einmal geändert. Wir verwenden von nun an den informellen Namen »Bell Labs«.

- Die Forscher konnten experimentieren und Programme nach Bedarf ändern. Da die Benutzerbasis nur sehr klein war, stellte es im Allgemeinen kein Problem dar, ein Programm notfalls von Grund auf neu zu schreiben. Und da die Benutzer gleichzeitig die Entwickler waren, konnten sie Probleme beheben, wenn sie auftraten, und Verbesserungen anbringen, wenn sich die Notwendigkeit dafür ergab.

  Unix selbst durchlief mehrere Versionen, die informell mit dem Buchstaben »V« und einer Ziffer bezeichnet werden: V6, V7 und so weiter. (Der formale Name folgte der Nummer des veröffentlichten Handbuchs: First Edition (erste Ausgabe), Second Edition (zweite Ausgabe) und so weiter. Zwischen den Namen besteht ein direkter Zusammenhang: V6 = Sixth Edition und V7 = Seventh Edition. Ebenso wie die meisten erfahrenen Unix-Programmierer verwenden wir beide Nomenklaturen.) Das einflussreichste Unix-System war die im Jahre 1979 veröffentlichte siebente Ausgabe (Seventh Edition; V7), obwohl Bildungseinrichtungen bereits seit mehreren Jahren frühere Versionen zur Verfügung standen. V7 zeichnete sich vor allem dadurch aus, dass es sowohl `awk` als auch die Bourne-Shell einführte, auf der die POSIX-Shell beruht. Etwa zu diesem Zeitpunkt erschienen auch die ersten Bücher über Unix.
- Die Forscher an den Bell Labs waren gut ausgebildete Informatiker. Sie entwarfen das System für ihren persönlichen Gebrauch sowie für ihre Kollegen, die ebenfalls Informatiker waren. Dies führte zu einem »Kein Nonsens«-Designansatz: Programme taten das, was man ihnen sagte, ohne übermäßig geschwätzig zu sein und mit Abfragen der Art »Sind Sie sicher?« zu nerven.
- Neben der einfachen Erweiterung des aktuellen Standes der Technik spielte vor allem eine Rolle, dass der Entwurf und das Lösen von Problemen elegant vonstatten gehen. Eine liebevolle Definition von Eleganz lautet »Kraft gepaart mit Einfachheit«.[2] Die Freiräume der Bell-Labs-Umgebung führten zu einem *eleganten* System, nicht nur zu einem *funktionalen*.

Diese Freiräume brachten natürlich auch einige Nachteile mit sich, die deutlich wurden, als Unix sich über seine Entwicklungsumgebung hinaus verbreitete:

- Zwischen den Hilfsprogrammen gab es viele Inkonsistenzen. Beispielsweise verwendeten Programme den gleichen Buchstaben für eine Option, diese hatten jedoch unterschiedliche Bedeutungen, oder sie verwendeten unterschiedliche Buchstaben für die gleiche Aufgabe. Die von unterschiedlichen Programmen verwendeten Syntaxen für reguläre Ausdrücke waren ähnlich, aber nicht identisch. Dies führte zu einer Verwirrung, die normalerweise hätte vermieden werden können. (Wäre ihre Bedeutung erkannt worden, dann hätten die Einrichtungen zur Filterung regulärer Ausdrücke in einer Standardbibliothek codiert werden können.)

2 Ich habe diese Definition das erste Mal in den 80er Jahren von Dan Forsyth gehört.

- Viele Hilfsprogramme wiesen Einschränkungen auf, etwa bezüglich der Länge der Eingabezeilen oder der Anzahl der offenen Dateien usw. (In modernen Systemen wurden diese Mängel im Allgemeinen behoben.)
- Manchmal wurden Programme nicht so gründlich getestet, wie es erforderlich gewesen wäre, wodurch es möglich wurde, sie versehentlich zum Absturz zu bringen. Dies führte zu überraschenden und verwirrenden »Core Dumps«. Glücklicherweise kommt dies bei modernen Unix-Systemen nur noch selten vor.
- Die Dokumentation des Systems war zwar im Allgemeinen vollständig, allerdings auch sehr knapp und minimalistisch. Es war daher schwieriger, sich in das System einzuarbeiten, als es eigentlich wünschenswert gewesen wäre.[3]

Der größte Teil dessen, was wir in diesem Buch präsentieren, dreht sich um die Verarbeitung und Manipulation *textueller*, nicht binärer Daten. Dies rührt aus dem starken Interesse an der Textverarbeitung her, die in den frühen Tagen von Unix existierte. Er ist aber auch aus anderen Gründen (auf die wir in Kürze eingehen werden) nützlich. Um genau zu sein, erfolgte der erste ernsthafte Einsatz eines Unix-Systems bei der Textverarbeitung und -formatierung in der Patentabteilung der Bell Labs.

Die ursprünglichen Unix-Maschinen (PDP-11-Maschinen der Firma Digital Equipment Corporation bzw. DEC) waren nicht in der Lage, große Programme auszuführen. Um eine komplexe Aufgabe zu bewältigen, musste man diese in kleinere Aufgaben unterteilen und jeweils von einem eigenen Programm erledigen lassen. Bestimmte allgemeine Aufgaben (das Extrahieren von Feldern aus Zeilen, das Ersetzen von Text usw.) kamen in vielen größeren Projekten häufiger vor, so dass dafür Standardwerkzeuge entwickelt wurden. Dies wurde schließlich als Vorteil erkannt: Das Fehlen eines großen Adressraums führte zu kleineren, einfacheren und *spezialisierteren* Programmen.

Viele Leute arbeiteten halbunabhängig an Unix, indem sie Programme anderer neu implementierten. Viele allgemeine Werkzeuge entwickelten sich in den verschiedenen Versionen auseinander; es bestand kein Grund für eine Standardisierung. Wird beispielsweise `grep` auf einem System mit der Option `-i` verwendet, bedeutet dies: »Ignoriere bei der Suche die Groß- und Kleinschreibung.« Auf einem anderen System wird dies mit der Option `-y` erreicht! Das ist bei vielen Hilfsprogrammen der Fall, nicht nur bei einigen. Die gebräuchlichen kleinen Hilfsprogramme wurden gleich benannt, allerdings würden die Shell-Programme, die für die Hilfsprogramme in einer Unix-Version geschrieben wurden, ohne Anpassung gar nicht auf einer anderen Unix-Version laufen.

Schließlich wurde es immer offensichtlicher, dass man einen allgemeinen Satz standardisierter Werkzeuge und Optionen brauchte. Das Ergebnis dieser Erkenntnis waren die POSIX-Standards. Der aktuelle Standard, IEEE Std. 1003.1–2004, umfasst sowohl die C-Bibliothek als auch die Shell-Sprache und die Systemprogramme sowie deren Optionen.

3 Das Handbuch bestand aus zwei Komponenten: der Referenz und dem Benutzerhandbuch. Das Letztere bestand aus Tutorials zu den meisten Teilen des Systems. Es war zwar möglich, Unix zu lernen, indem man die ganze Dokumentation durchlas, und viele Leute (einschließlich der Autoren) taten auch genau das, allerdings werden die Systeme heutzutage nicht mehr mit gedruckten Dokumentationen dieser Art geliefert.

Das Gute ist, dass sich die Standardisierungsanstrengungen bereits ausgezahlt haben. Moderne kommerzielle Unix-Systeme folgen ebenso wie frei verfügbare Versionen wie GNU/Linux und BSD-Derivate dem POSIX-Standard. Das Lernen von Unix vereinfacht sich dadurch, außerdem können portable Shell-Skripten geschrieben werden. (Beachten Sie jedoch Kapitel 14.)

Interessanterweise war POSIX nicht der einzige Versuch der Unix-Standardisierung. Vor allem eine anfangs europäische Gruppe von Computer-Herstellern namens X/Open stellte ihren eigenen Satz von Standards her. Der bekannteste war XPG4 (X/Open Portability Guide, vierte Ausgabe), der zuerst im Jahre 1988 erschien. Es gab auch ein XPG5, bekannter als Unix 98-Standard oder als *Single Unix Specification*. XPG5 enthielt größtenteils POSIX als Teilmenge und hatte ebenfalls einen gewissen Einfluss.[4]

Die XPG-Standards waren vielleicht weniger streng in ihrer Sprache, deckten aber eine breitere Basis ab, wodurch sie einen größeren Bereich der existierenden Praxis bei Unix-Systemen dokumentierten. (Das Ziel bestand für POSIX darin, einen Standard zu gestalten, der so formell war, dass er als Richtlinie für eine völlige Neuimplementierung – selbst auf Nicht-Unix-Plattformen – verwendet werden konnte. Dadurch wurden schließlich viele Funktionen und Eigenschaften, die auf Unix-Systemen verbreitet waren, zunächst aus den POSIX-Standards ausgeschlossen.) Der POSIX-Standard von 2001 fungiert gleichzeitig als XPG6, indem er die *X/Open System Interface Extension* (oder kurz XSI) einschließt. Dabei handelt es sich um eine förmliche Erweiterung des zugrunde liegenden POSIX-Standards, die Attribute dokumentiert, mit denen ein System nicht nur POSIX-konform, sondern auch XSI-konform wird. Das bedeutet, es gibt inzwischen nur noch ein formelles Standarddokument, auf das sich System- und Anwendungsprogrammierer beziehen müssen. (Es dürfte kaum überraschen, dass dieses als Single Unix Standard bezeichnet wird.)

In diesem Buch konzentrieren wir uns auf die Shell-Sprache und die Unix-Hilfsprogramme, die durch den POSIX-Standard definiert werden. An den entsprechenden Stellen weisen wir auch auf Funktionen hin, die XSI-spezifisch sind, da Sie diese wahrscheinlich auch benutzen werden.

1.2 Prinzipien für Software-Werkzeuge

Im Laufe der Zeit wurde eine Reihe von Prinzipien für den Entwurf und das Schreiben von Software-Werkzeugen entwickelt. Sie werden diese Prinzipien in den Programmen veranschaulicht sehen, die in diesem Buch vorgestellt werden. Gute Software-Werkzeuge sollten folgenden Richtlinien genügen:

4 Die Liste der X/Open-Publikationen steht unter *http://www.opengroup.org/publications/catalog/* zur Verfügung.

Mache eine Sache gut

Dies ist in vielerlei Hinsicht das wichtigste zu erfüllende Prinzip. Programme, die nur für eine Aufgabe zuständig sind, sind leichter zu entwerfen, leichter zu schreiben, leichter zu reparieren und leichter zu warten sowie zu dokumentieren. Zum Beispiel sollte man von einem Programm wie `grep`, das Dateien nach Zeilen durchsucht, die einem Muster entsprechen, *nicht* erwarten, dass es auch noch Berechnungen ausführt.

Eine natürliche Folge dieses Prinzips ist eine starke Zunahme kleinerer, spezialisierter Programme, so wie ja auch ein professioneller Zimmermann eine große Anzahl von Spezialwerkzeugen in seinem Werkzeugkoffer hat.

Verarbeite Textzeilen, keine binären Zeilen

Textzeilen bilden in Unix das universelle Format. Datendateien, die Textzeilen enthalten, sind einfach zu verarbeiten, wenn Sie Ihre eigenen Werkzeuge schreiben, sie sind mit jedem verfügbaren Texteditor leicht zu verändern, und sie können über Netzwerkgrenzen und unterschiedliche Rechnerarchitekturen hinweg übertragen werden. Die Benutzung von Textdateien erleichtert die Kombination eigener Werkzeuge mit existierenden Unix-Programmen.

Benutze reguläre Ausdrücke

Reguläre Ausdrücke stellen einen leistungsfähigen Mechanismus für das Arbeiten mit Text dar. Wenn Sie verstehen, wie sie funktionieren und wie Sie sie richtig einsetzen können, dann erleichtern Sie sich das Schreiben von Skripten.

Obwohl reguläre Ausdrücke sich im Laufe der Jahre bei den verschiedenen Werkzeugen und Unix-Versionen unterschiedlich entwickelt haben, bietet der POSIX-Standard nur zwei Arten von regulären Ausdrücken mit standardisierten Bibliotheksroutinen für die Filterung von regulären Ausdrücken. Damit können Sie Ihre eigenen Werkzeuge schreiben, die mit regulären Ausdrücken, identisch mit denen von `grep` (von POSIX *Basic Regular Expressions* oder BREs genannt) oder mit denen von `egrep` (von POSIX *Extended Regular Expressions* oder EREs genannt), funktionieren.

Verwende immer die Standardein- oder -ausgabe

Wenn nicht ausdrücklich ein Dateiname angegeben wurde, mit dem gearbeitet werden soll, dann soll ein Programm die Daten von seiner Standardeingabe lesen und Daten an seine Standardausgabe schreiben. Fehlermeldungen sollen immer an die Standardfehlerausgabe gehen. (Diese werden in Kapitel 2 besprochen.) Werden Programme auf diese Art geschrieben, können sie leicht als Daten*filter* verwendet werden – d.h. als Komponenten in größeren, komplizierteren Pipelines oder Skripten.

Sei nicht geschwätzig

Software-Werkzeuge sollten nicht »geschwätzig« sein. Meldungen der Art `beginne die Verarbeitung`, `bin fast fertig` oder `Verarbeitung abgeschlossen` sollen nicht mit der normalen Ausgabe eines Programms vermischt werden (oder zumindest nicht standardmäßig).

Wenn Sie bedenken, dass Werkzeuge in einer Pipeline aneinander gehängt werden können, dann ist das sinnvoll:

```
werkzeug_1 < datendatei | werkzeug_2 | werkzeug_3 | werkzeug_4 > ergebnisdatei
```

Würde jedes Werkzeug Meldungen wie »Ja, ich arbeite« erzeugen und in die Pipe schicken, dann würden die Daten, die verarbeitet werden sollen, hoffnungslos zerstört werden. Selbst wenn die einzelnen Werkzeuge ihre Nachrichten an die Standardfehlerausgabe schicken würden, wäre der Bildschirm voll mit sinnlosen Fortschrittsmeldungen. Bei Werkzeugen gilt immer: Keine Nachrichten sind gute Nachrichten.

Dieses Prinzip hat noch eine weitere Implikation. Im Allgemeinen folgen Unix-Werkzeuge der Philosophie »Du hast darum gebeten, du bekommst es auch«. Sie fragen nicht noch einmal »Bist du wirklich sicher?«. Wenn ein Benutzer `rm irgendeinedatei` eintippt, gehen die Unix-Entwickler davon aus, dass er weiß, was er tut. `rm` löscht also die Datei, ohne noch einmal nachzufragen.[5]

Erzeuge dasselbe Ausgabeformat, das auch als Eingabe akzeptiert wurde

Spezielle Werkzeuge, die erwarten, dass die Eingabe einem bestimmten Format genügt – etwa Header-Zeilen, gefolgt von Datenzeilen, oder Zeilen mit bestimmten Feldtrennern usw. –, sollten eine Ausgabe erzeugen, die den gleichen Regeln gehorcht wie die Eingabe. Dies macht es einfach, die Ergebnisse eines Programmdurchlaufs durch einen anderen Programmdurchlauf, möglicherweise mit anderen Optionen, zu verarbeiten.

Beispielsweise manipuliert die `netpbm`-Programmsuite[6] Bilddateien, die im Portable-Bitmap-Format gespeichert sind.[7] Diese Dateien enthalten Bitmap-Bilder, die mit Hilfe eines wohldefinierten Formats beschrieben werden. Jedes Werkzeug liest PBM-Dateien, verändert das enthaltene Bild auf irgendeine Weise und schreibt dann wieder eine Datei im PBM-Format aus. Dadurch können Sie leicht eine Pipeline konstruieren, mit der komplizierte Bildverarbeitungen durchgeführt werden, etwa das Skalieren eines Bildes mit anschließendem Drehen und Verringern der Farbtiefe.

Lass jemand anderes den schwierigen Teil erledigen

Oft ist es so, dass es kein Unix-Programm gibt, das *genau* das tut, was Sie brauchen. Es ist jedoch möglich, 90 Prozent der Arbeit von existierenden Werkzeugen erledigen zu lassen. Sie können dann, falls erforderlich, ein kleines, spezialisiertes Programm schreiben, um die Aufgabe abzuschließen. Auf diese Weise ersparen Sie sich eine Menge Arbeit und müssen nicht alle Probleme jedes Mal von Grund auf neu lösen.

---

5 Falls Ihnen das zu riskant ist, können Sie die Option `-i` mit `rm` verwenden. Diese zwingt `rm`, nach einer Bestätigung zu fragen. Auf jeden Fall bittet `rm` vor dem Löschen verdächtiger Dateien um eine Bestätigung, etwa bei Dateien, deren Berechtigungen das Schreiben verbieten. Wie immer gibt es einen Mittelweg zwischen den Extremen, niemals nachzufragen und immer nachzufragen.

6 Die Programme sind nicht standardmäßig Bestandteil des Unix-Toolsets, werden aber auf GNU/Linux- und BSD-Systemen üblicherweise installiert. Der Ausgangspunkt im Web ist *http://netpbm.sourceforge.net/*. Folgen Sie von dort aus den Verweisen auf die Sourceforge-Projektseite, die wiederum Verweise zum Herunterladen des Quellcodes enthält.

7 Es gibt drei unterschiedliche Formate; siehe die *pnm*(5)-Manpage, falls `netpbm` auf Ihrem System installiert ist.